Leveraging Kernel Security Mechanisms to Improve Container Security: a Survey

Maxime Belair, Sylvie Laniepce, Jean-Marc Menaud

Leveraging Kernel Security Mechanisms to Improve Container Security: a Survey

Maxime Bélair
Orange Labs
IMT Atlantique
Caen, France
maxime.belair@orange.com

Sylvie Laniepce
Orange Labs
Caen, France
s.laniepce@orange.com

Jean-Marc Menaud
IMT Atlantique
STACK, INRIA, LS2N
Nantes, France
jean-marc.menaud@imt-atlantique.fr

ABSTRACT

Containerization is a lightweight virtualization technique reducing virtualization overhead and deployment latency compared to full VM; its popularity is quickly increasing.

However, due to kernel sharing, containers provide less isolation than full VM. Thus, a compromised container may break out of its isolated context and gain root access to the host server. This is a huge concern, especially in multi-tenant cloud environments where we can find running on a single server containers serving very different purposes, such as banking microservices, compute nodes or honeypots. Thus, containers with specific security needs should be able to tune their own security level.

Because OS-level defense approaches inherited from time-sharing OS generally requires administrator rights and aim to protect the entire system, they are not fully suitable to protect usermode containers. Research recently made several contributions to deliver enhanced security to containers from host OS level to (partially) solve these challenges.

In this survey, we propose a new taxonomy on container defense at the infrastructure level with a particular focus on the virtualization boundary, where interactions between kernel and containers take place. We then classify the most promising defense frameworks into these categories.

CCS CONCEPTS

• **Security and privacy → Virtualization and security**; *Access control*; • **Computer systems organization → Cloud computing**; • **General and reference →** *Surveys and overviews.*

KEYWORDS

Container, Security, Virtualization, Virtualization Boundary, LSM, NFV

1 INTRODUCTION

Containerization is a lightweight virtualization technique in which containers are virtual domains offering usermode execution context, while sharing the host kernel at the host level. Containers are isolated from each other and from the host. Containers interact with the host kernel by sending system calls (syscalls) through the virtualization boundary. Compared to full virtualization, containerization reduces virtualization overhead and resource usage, offers reduced deployment latency and finally improves reusability [8]. For these reasons, the popularity of containers in cloud-based environments is quickly increasing with popular tools such as LXC [21], Docker [23] or Kubernetes [14].

On the other hand, due to kernel sharing, containers have a greater attack surface than full VMs thus provide less isolation. Therefore, a compromised container may break out of its isolated context and gain access to the host server. For instance, the recent vulnerability CVE-2019-5736 [25][12] lets a maliciously crafted container arbitrarily rewrite the containerization's core software (runC, used in most containerization engines) enabling to get full root access over the host, incidentally taking control of the other containers collocated on the node and providing an entry point in the DeMilitarized Zone (DMZ) for further attacks. This risk is also increased by the fact that untrustworthy containers can be downloaded and executed directly from the internet (e.g. Docker Hub). In multi-tenant cloud environments, we can find running on a single server containers serving very different purposes, such as banking microservices, webservers, compute nodes or honeypots. Since companies commonly use thousands or even millions of containers simultaneously, manual defense approaches provided by the administrator are not very practical.

For these reasons, containers with specific OS-level security needs should be able to tune their own security level. But this is not currently possible because OS-level security frameworks generally apply to the whole system meaning that security cannot be applied only to a single container. Because containers are designed to run without administrator rights, they cannot use the numerous already existing tools that require administrator privileges. Finally, because container design differs from full VM or Unix process, related defense approaches are often unsuitable for containers.

Several recent defense approaches try to tackle these issues by leveraging cooperation between the container and the host. Because all the interactions required to setup defense mechanisms must go through the virtualization boundary, it is an interesting point to look at to design security primitives. Therefore, in this survey, we propose a new taxonomy for container OS-level defense at infrastructure level with a strong focus on data transmitted through the virtualization boundary.

If previous studies already featured guides for container defense [34], to the best of our knowledge no study featured a clear container defense taxonomy and especially with the focus of the data transmitted through the virtualization boundary.

In the remainder of this paper, we further present containerization and associated security challenges (2), we then propose a new taxonomy and we classify some of the most promising defense approaches (3). We explore container security in a use case : Network Function Virtualization (4) and we finally discuss future research directions on container security managed by the host (5).

2 BACKGROUND: CONTAINERS AND ISOLATION

While modern OSes provide a good level of hardware abstraction and isolation (e.g. a process cannot see the memory of other processes), the level of software isolation remains low, even for non-root processes (e.g. a process can have information about other processes with commands like ps).

Virtualization is an old technique that improves the level of software isolation of a process (or a set of processes) from other virtualized processes. In early implementations, the whole OS was virtualized, sometimes with extra features such as hypercalls or optimized drivers.

Containerization is a more recent approach than full virtualization providing OS-level virtualization that is, userspace virtualization while abstracting the kernel. In some regards, containers can be seen as a successor of chroot [9], a first attempt to isolate processes. Container's isolation is provided by Linux namespaces [26] (providing isolation for pid, mount points, network, ...). Thus a container can be configured to finely sandbox an environment.

In the remaining of this section, we present Linux Security Modules (LSM), a powerful defense framework targeting Linux and the challenges faced by container security at OS-level including why tools like LSM cannot be easily adapted to containers to improve their security.

2.1 Linux Security Modules

LSM [36] is a kernel-level security framework initially targeting Linux (i.e. not containers). LSM composes the standard approach to provide orthogonal access control models [30] to Unix's Discretionary Access Control (DAC). LSM design is very flexible and can offer a broad range of security types. These controls are implemented in LSM modules which are built above the LSM framework. Due to their common base, these modules use the same primitives and can be used interchangeably at boot time, although currently, only one major module can be used by the OS, loaded at boot-time. Some of the most widely used LSM modules are:

- **SELinux** [33] provides a file labeling system enabling Mandatory Access Control (MAC) or Role-Based Access Control (RBAC) usage to finely enforce policies. This framework is powerful and flexible yet complex to use. Because labeling has to be done manually, SELinux is not really adapted to cloud environments;
- **AppArmor** [4] is another alternative to provide Mandatory Access Control (MAC) or Role Based access control (RBAC). AppArmor confines individual programs to a set of files, capabilities, network accesses and rlimits. AppArmor is often considered easier to use than SELinux;
- **Integrity Measurement Architecture (IMA)** [27] provides system integrity enforcement by calculating hash values of specific files and verifying their adherence to expected values. These controls are enforced by a Trusted Platform Module (TPM), a hardware chip widely deployed on modern processors. However, TPM has a limited number of registers, it cannot control an arbitrary number of containers.

In its current design, LSM usage and policies enforcement can only be handled by the system administrator.

Technically, the LSM framework provides a convenient set of kernel hooks placed at strategical points. Modules subscribe to these hooks allowing them to execute security code just before an operation of interest is made by a process, such as a file creation or a network packet reception. Thus, LSM modules can take appropriate actions matching policies defined for this particular operation. Because modules are built over the LSM framework, they can dereference kernel pointers (e.g. the second argument of the syscall 'open' is the address of the file path to open) giving them the visibility needed to finely enforce policies. Due to the efficiency of the LSM approach, research is in progress to adapt it to containers, as further detailed in section 3 of this survey.

2.2 Container security

As stated above, partially because of the weaker isolation of containers than full-VM environments, containers with specific OS-level security needs should be able to tune their own security level. But, while numerous OS-level or full-VM defense frameworks already exist, they are generally not suitable for container defense due to these limitations [35]:

- **Mandatory**: Administrator rights are required to use these frameworks. Because under normal circumstances, containers do not have administrator rights over the host, they cannot use of the numerous security techniques that require it;
- **Global**: As these frameworks apply to the whole system, they cannot be used to protect only a subset of the system (such as a single container);
- **Unsuitable for container architecture**: By design, container security faces different challenges than full-VM security. A main challenge in full virtualization environments is that the hypervisor does not always know the guest's semantics, thus cannot easily provide security. This problem, known as semantic gap [17], is not an issue here as containers share their semantics with the host. Lastly, container adoption is partially due to its good performance compared to full VMs. A significant part of traditional techniques incurs a non-negligible overhead and is therefore not adapted for container defense.

These concerns are particularly evident in LSM modules that cannot be easily adapted to containers due to their mandatory and global design. Additionally, if namespacing the LSM framework itself seems very appealing because it would overcome above concerns and would allow a container to safely use alternate LSM modules, research showed that in its current design, namespacing LSM at framework level is a very hard challenge [20] mostly because of the lack of semantics information at framework level and of the use of global variables for networking (e.g. secids, secmark, ...). Due to this limitation at framework level, LSM modules have to handle namespacing by themselves to be useful for container security. Currently, if namespacing is already implemented in AppArmor, allowing both a container and its host to use AppArmor simultaneously, this feature is still under discussion or development for other LSM modules. Therefore, to the best of our knowledge, every adaptation attempt of LSM up to now is either specific to a single module, *ad hoc* or makes the framework only available to the

host. As detailed in the remaining of this paper, the goal of enabling containers to transparently use all LSM modules on their own is not yet reached.

3 NEW TAXONOMY FOR CONTAINER DEFENSE

In this section, we describe some approaches for the infrastructure (host OS) to improve the security level of a container. We classify these techniques regarding how the data required to setup and enforce security (if any) is transmitted between the container and the host-kernel.

- **Configuration-based defense**: Policies are defined and enforced solely by the host as a configuration file or string;
- **Code-based defense**: The container pushes code in the host kernel to enforce policies. The kernel executes it when the container makes an operation related to these policies;
- **Rule-based defense**: The container pushes string rules to the kernel. They are interpreted and enforced by a handler in the host kernel.

We argue that this taxonomy is complete because either no data is transmitted from the container to the host, and then the defense mechanism is configuration-based, or data is transmitted. In the latter case, either the pushed data is code or not in which case data can be seen as rules.

In the remaining of this section, we further detail these categories. We claim that this categorization clearly underlines the interests and the inherent limitations of these techniques. We also classify some of the main defense frameworks.

3.1 Configuration-based defense

The Configuration-based approaches enable the host to customize with a configuration file the properties of a container such as its network interfaces or its mounted directories. Used in most containerization engines, this method is flexible and easily usable from the host. A well-configured container has significantly improved security properties. For instance, Docker provides a way to tune containers' isolation with appropriate command line interface arguments or within the DockerFile. This technique is also heavily used by containers orchestrators. For instance, Marathon [24] provides a REST API to manage containers including their security. However, because this technique is static and must be enforced by the host, there is no generic way for a container to handle or update its own configuration. Thus, a container cannot use this approach to enforce its own security policies. Additionally, if this technique may be efficient for a well-known container, it is way more limited to protect an arbitrary container whose behavior is unknown.

Containerization engines already provide ways to use some Linux Security primitives for containers through configuration files or strings. Containerization engines also provide a way to customize rights more finely than root/non-root with capabilities such as "create a port under 1024" or "use the setuid syscall". Efforts are made to further protect individual containers by allowing the host to apply customized LSM modules' policies. For instance, AppArmor [4] or Tomoyo [15] security profiles can be attached to a container and similar work is in progress for other LSMs. These measures provide a first step to container security by enforcing isolation and restricting the attack surface. But this protection remains limited due to i) the restricted number of features available, ii) the fact that only the system administrator can use configuration-based approaches and iii) their static nature (policies are enforced at creation time and cannot be modified by the container).

DIVE [5] is a very recent approach allowing an orchestrator to check the integrity of a given container on a remote compute node. DIVE relies on a modified version of IMA [27] supporting containers and a remote attestation framework (OpenAttestation). Container's integrity is checked at host level, that is without any interaction with the container. If a container is compromised, the orchestrator can request the rebuild of this single container (meaning the whole system). Therefore, with the help of DIVE, the infrastructure can periodically check that the hosted containers are not compromised. Yet, since DIVE only monitors files' integrity, it cannot detect any malicious in-memory modifications. As a significant part of attacks, for instance buffer overflows, can lead to in-memory modifications, DIVE in its current form does not satisfy all the security needs of containers.

Cilium [19] is another framework for network security by making a level 7 proxy usable to further confine containers. It enables to tune for each class of container which access points of a Rest API are accessible. This is particularly useful in microservices environments where a malicious access to poorly designed APIs could lead to attacks such as confused deputy [16]. Because Cilium uses eBPF (extended Berkeley Packet Filter) and can work with XDP (eXpress Data Path), it can improve packet throughput compared to iptables. Yet, Cilium's focus is very tight, so it must be associated with other frameworks to significantly improve security.

3.2 Code-based defense

The Code-based approaches enable containers to push code to the host kernel. When the container performs a monitored action on a given object, the container-pushed code is executed to verify the operation's legality. Therefore, this flexible technique allows a container to implement and enforce its own security logic. This technique is especially promising in multi-tenant cloud environments where containers' security needs can be very dissimilar. Because this approach allows a sensitive container to enforce its own policies by defining and pushing code controlling its security to the kernel, it mitigates potential compromises without relying on external tools (which would need to be installed system-wide by the administrator). The most known code pushing technique, called eBPF [2] is a standard feature of Linux Kernel and is considered as relatively mature. Because the kernel can check the innocuousness of the code pushed by the container but does not modify it, this technique cannot be customized by the host to provide security. However, this technique allows code provided by an untrusted environment to be executed directly inside the host's kernel which could potentially lead to vulnerabilities or Denial of Service (DoS) if not properly sanitized. The sanitization process reduces the possibilities of eBPF code by restricting usable code, forbidding loops, limiting the code size, ... Due to these limitations, very complex inputs cannot be parsed and handled by eBPF. Finally, writing correct and fast defense code can be complex for a standard container user (helpers can sometimes abstract the code generation).

Seccomp-BPF [11] is a sandbox mechanism integrated to the Linux Kernel that permits to tighten container's attack surface by applying policies to syscalls based on eBPF code. But since this code cannot dereference syscall kernel pointers, this filtering remains too coarse grained to significantly improve container's security.

Landlock LSM [28] has recently been proposed to let a non-administrative process (e.g. a container) finely sandbox itself to limit its own rights thus mitigating potential compromises. Landlock limits Trusted Computing Base (TCB) modifications using mature and trusted software (LSM, eBPF) with few modifications. Landlock is designed to be stackable with other LSM modules and mainly provides to unprivileged containers files access control. At high level, Landlock provides a way for non-root containers to push eBPF code in the host kernel, like in seccomp-bpf, but as Landlock is based on LSM framework, it can dereference kernel pointers, thus enabling more precise access restriction. Technically, Landlock setups maps containing the policies to enforce, for instance the access control policy applicable to a particular directory. It then pushes the code in charge of enforcing these policies to the right hooks in the kernel. Thus, at execution, when a monitored operation is executed (e.g open a file), the eBPF code checks its compliance. Yet, Landlock suffers from drawbacks. First, it can only enforce integrity and policies on files (as network cannot be mounted dynamically without administrative rights). Second, the current implementation (v8) requires administrative rights [29] to create eBPF maps, that limits its interest in real-world scenarios. Finally, writing eBPF program for Landlock can sometimes be complex although Landlock provides an abstraction layer to apply read-write, read-only and deny policies for files. Therefore, Landlock does not satisfy all the security needs of containers on itself.

3.3 Rule-based defense

The Rule-based approaches enable containers to push string rules to the host, which are interpreted and enforced by the host kernel. Policies can be expressed as standard strings (e.g. {file:"/etc/passwd", expected:"0xF2A72CC5"}) more easily writable than eBPF code. Therefore, with a such technique, cloud operators keep some control over container security because they can manage the set of security tools a container can interact with. As the host proactively enforces these policies, it can check their legality and their innocuousness at run-time. On the other side, this technique is less flexible than code-based defense because the addition of a new security feature requires kernel recompilation by the administrator. Thus, a container with specific security needs may not have the tools it needs to fulfill its security needs and cannot add its custom policies.

Pledge [6] provides a simple sandboxing mechanism to restrict the behavior of processes (possibly a container), thus mitigating potential compromises. Unlike seccomp-bpf, pledge is based on the allowance of a set of behaviors and not syscalls (e.g. stdio for libc access, rpath for read-only access to files...). These policies are enforced with the help of a new syscall (pledge). Kernel hooks enable to control the syscall legality at usage. Pledge can be called several times to further reduce abilities (because the rights at usage can often be lower than at initialization). Yet, Pledge behavior remains coarse grained and this framework is very specific to OpenBSD.

Security Namespace [35] is a recent proposal aiming to enable containers to use LSM modules to improve their own security. Security Namespace does not provide a generic model to adapt LSM modules to containers (which would be difficult since these modules differ greatly in design and details) but offers a "manual" adaptation of some of them (IMA, Apparmor) and aims at providing a model to adapt the others. In order to do that, a new 'Security' namespace abstraction is defined in addition to the seven already existing namespaces. Security Namespace relies on the same mechanisms as the others namespaces (e.g. a flag on syscalls clone, fork or unshare for creation) so it can be used transparently by containers. A key feature of Security Namespace is the automatic detection and management of inconsistent policies about any file (e.g. read-only vs read-write access) which could lead to a false feeling of security or to a Denial of Service. At system initialization, LSM hooks handlers are installed with an empty list of policies to handle. Policies can then be pushed as string rules from the container to the framework and are appended to the list in the kernel. When an operation is made over a resource, every Security Namespace that has visibility over it checks the conformity to the policy. The access is granted only when allowed by all these Security Namespaces. Because Security Namespace requires intensive modifications over kernel base and the LSM modules to use these modules and works with only a subset of LSM modules (i.e. no genericity), it cannot be seen as a general solution to container security in its current implementation.

3.4 Recap chart

Figure 1 provides a comparison of the security frameworks above presented.

	Based on Config. (C), Code (X) or Rule (R)	Granularity	Customizability	Usable without software modifications	Genericity and integration	Scope
Config. Tools	C	●	○	●	●	●
DIVE [5]	C	○	○	●	◐	○
Cilium [19]	C	●	○	●	◐	○
Seccomp-BPF [11]	X	○	◐	○	●	◐
Landlock [28]	X	●	●	●	○	●
Pledge [6]	R	○	○	○	◐	◐
Security NS [35]	R	●	○	●	○	●

Figure 1: Comparison of security frameworks

The first column shows whether the framework is configuration-based (C), code-based (X) or rule-based (R). We remind that configuration-based approaches allow the host to define and handle containers' protection on its own while code-based and rule-based approaches allows the container to initiate a "demand" of protection, fulfilled by the host kernel. The second column shows whether

the technique provides fine or coarse grained security (e.g. for a framework monitoring syscalls: whether it can restrict syscalls to precisely match a defined behavior). Customizability column (3) shows whether the approach can provide further protection services that it was not explicitly designed for. Because Landlock and Seccomp-BPF enable (within some restrictions) to push any valid BPF code to the host, thus, they can provide a further customized security service while other approaches have a fixed set of features. The following column (4) shows whether policies are defined directly in the software to be protected (e.g. the task running in a container), meaning source code modifications are required to enforce policies in this software. Because Pledge and Seccomp-BPF provide mechanisms for a process to confine itself while other frameworks confine a full environment, modifications have to be made directly in the source code of this software and thus are less suitable to protect closed-source software. Genericity and integration column (5) shows whether the framework has been designed in a generic way, is well integrated into production environments and is usable in real life scenarios. For instance, in its current version, Landlock actually requires administrator rights to work thus limiting its usability in production where containers run in user-mode. Overall, despite no solution provides a general solution to all security concerns related to containers, some frameworks cover a broader scope than others, as shown in the last column (6).

The following section illustrates why leveraging techniques presented in this survey can be needed for containers and how they can help to improve their security level in a use case, namely Telecom's Network Function Virtualization (NFV).

4 USE CASE: CONTAINER-BASED NETWORK FUNCTION VIRTUALIZATION

Telecom networks used to contain a broad range of proprietary hardware appliances such as routers, firewalls or carrier-grade NAT (Network Address Translation) that need to be deployed, operated and updated physically, leading to complexities in the architecture. NFV is an inexorable trend consisting in virtualizing and centralizing these appliances into industry standard compute servers to simplify these architectures, reduce costs and improve overall reliability. In such a scenario, each appliance is virtualized into a (set of) full VM(s) specialized in a specific task. But, due to container's greater performance and flexibility, containerization receives more and more attention to virtualize network functions in research environments.

Nevertheless, because data handled in NFV environments come directly from untrusted and potentially malicious users, NFV security remains not perfect and any compromise may have a strong impact on the operator's infrastructure and incidentally on users (e.g. the theft of users' network history). Due to these concerns and because of NFV's restrictive standards, the use of containers remains very limited in production environments.

Operators should therefore leverage additional security measures to get enough confidence in their architecture to safely use containers and take advantage of their improved properties. In NFV environments, Telecom operators operate their own infrastructure with administrator rights (unlike in multi-tenant cloud environments). Therefore, with NFV, they have a higher flexibility to use cutting-edge security frameworks including these presented in this paper, even these requiring administrator rights. Therefore, adapting container security to NFV environments seems to be a promising line of application.

As shown in this survey, defense can be initiated either by a container or by the host. We show here that both ways can be relevant to protect containerized NFV networks. It has to be understood that these two approaches are not exclusive and an operator willing to perform defense in depth should consider both.

NFV containers can be protected without interaction with the host using configuration-based tools (i.e. the security is initiated and enforced by the host). With such techniques, the infrastructure is able to finely tune the NFV isolation, integrity and access rights. This can be done not only by configuring namespaces and tuning DAC access, but also by integrating additional Access Control Models, filtering network accesses at OSI level three or seven, ... Configuring the defense of a well known and documented NFV is not a complex process since its behavior is known, it is possible to restrict NFV capacities to the minimum without altering its operation. This process can be more complex for greybox NFVs where configuration attempts have to be done "blindly" and could for instance block an exceptional but essential case resulting in an incorrect behavior and a false positive.

Additionally, NFV containers can proactively participate to their own defense (relying on security primitives delivered by the host) leveraging code-based or rule-based defense to improve their confidence in their security. With such an approach, containers can use on their own security tools (based on the LSM framework or not) exposed by the host. They can also execute an 'arbitrary' policy code with mechanisms such as eBPF thus achieving very high flexibility. Besides defense in depth, third parties sometimes deliver NFV as greybox containers. In this case, security policies have to be defined by the third-party because it would be challenging (and sometimes contractually forbidden) for an operator to reverse-engineer the greybox NFV to define security (the operator remains in charge of comprehensively testing the containers). Thus, although it can seem intuitive to use only host-defined policies to centralize security, this is not always possible nor desirable. For instance, a third party might want to be guaranteed that the security policies he defined are enforced. A way to achieve this is to enforce them from the container, therefore with code-based or rule-based techniques.

In conclusion, container security is a strong problematic for a broad range of industries, such as Telecom relying in NFV. Because container security approaches still have some limitations (as shown in section 3), research have to be made to overcome them.

5 DISCUSSION AND FUTURE WORKS

In this paper, we presented a new taxonomy for container security with a particular focus on data transmitted through the virtualization boundary. We showed that each type of defense offers distinct features and does not have the same limitations. We believe that our classification can help understanding and designing new security features for containers. To the best of our knowledge, this is the first study of its kind.

Containerization is still a relatively young and evolving technology but is widely used in cloud environments. However, its security is still in an early phase and faces unsolved challenges.

Making LSM usable by individual containers would significantly improve their security and is therefore a promising line of research. This remains a hard problem as LSM modules are very diverse, regarding their objectives, architecture and implementation details. Some research aim to design a new namespace to generically handle security. Another line of research which remains to be explored is to rethink LSM architecture to ease its integration with containers.

Some approaches aim to design hybrid abstractions with the advantages of both containers and Light VMs. For instance, gVisor [13] creates a new container abstraction in which a userspace kernel checks and proxies syscalls to the host kernel, improving security at the cost of size and performance overheads and reduced syscall compatibility. Lighter VMs and optimized hypervisors that are competitive in performance with containers while keeping better isolation and security properties are also proposed [1][10]. Finally, some approaches [22][32] create new abstractions usable like containers but internally using library OS [7], allowing greater performance and tightened attack surface but remain complex to use and often require to fully rewrite software.

Complementarily to kernel-level defense, some approaches aim to improve container security by interacting directly with the hardware. For instance, Scone [3] enables containers to deport critical code execution into a secure "enclave" leveraging Intel Software Guard eXtension (SGX) [18]. This mechanism allows to protect containers from a malicious host. Yet, Scone incurs a high overhead and requires non-trivial software modifications. Finally, due to critical vulnerabilities found in SGX [31] cloud operators may want to disable SGX, incidentally disabling Scone.

REFERENCES

[1] Amazon. 2019. AWS Firecracker GitHub Repository. https://github.com/firecracker-microvm/firecracker. (2019).

[2] Pratyush Anand. 2017. A presentation of eBPF. https://opensource.com/article/17/9/intro-ebpf. (2017).

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, and others. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 689–703.

[4] Mick Bauer. 2006. Paranoid Penguin: An Introduction to Novell AppArmor. *Linux J.* 2006, 148 (Aug. 2006), 13–.

[5] Marco De Benedictis and Antonio Lioy. 2019. Integrity verification of Docker containers for a lightweight cloud environment. *Future Generation Computer Systems* (2019).

[6] Theo De Raadt. 2015. pledge(), a new mitigation mechanism *(Hackfest '15)*. Québec. https://www.openbsd.org/papers/hackfest2015-pledge/mgp00001.html

[7] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. *SIGOPS Oper. Syst. Rev.* 29, 5 (Dec. 1995), 251–266.

[8] Wes Felter, Alexandre Ferreira, Ram Rajamony, and others. 2014. An Updated Performance Comparison of Virtual Machines and Linux Containers. *technology* 25 (2014), 31.

[9] Free Software Foundation. 2019. Chroot man page (2). http://man7.org/linux/man-pages/man2/chroot.2.html, (2019).

[10] OpenStack Foundation. 2019. Kata Containers Website. https://katacontainers.io/. (2019).

[11] freedesktop.org. 2017. Presentation of Seccomp BPF. https://dri.freedesktop.org/docs/drm/userspace-api/seccomp_filter.html. (2017).

[12] Nick Frichette. 2019. PoC for CVE-2019-5736-PoC. https://github.com/Frichetten/CVE-2019-5736-PoC. (2019).

[13] Google. 2019. GVisor GitHub repository. https://github.com/google/gvisor. (2019).

[14] Google. 2019. Kubernetes GitHub repository. https://github.com/kubernetes/kubernetes. (2019).

[15] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. 2004. Task Oriented Management Obviates Your Onus on Linux. *Linux Conference 2004* 3 (2004).

[16] Norm Hardy. 1988. The Confused Deputy: (or Why Capabilities Might Have Been Invented). *SIGOPS Oper. Syst. Rev.* 22, 4 (Oct. 1988), 36–38.

[17] Yacine Hebbal, Laniepce Sylvie, and Jean-Marc Menaud. 2015. Virtual Machine Introspection: Techniques and Applications. In *International Conference on Availability, Reliability and Security*. Toulouse, France. https://hal.inria.fr/hal-01165285

[18] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, and others. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*. ACM, New York, NY, USA, Article 11, 1 pages.

[19] Isovalent Inc. 2019. Cilium GitHub repository. https://github.com/cilium/cilium. (2019).

[20] Jhon Johansen. 2018. Making Linux Security Modules available to Containers: Stacking and Namespacing the LSM. In *Proceeding of the Free and Open Source software Developers' European Meeting (FOSDEM '18)*. Brussels. https://archive.fosdem.org/2018/schedule/event/containers_lsm/

[21] Daniel Lezcano, Stéphane Hallyn, and Graber Stéphane. 2018. LXC GitHub repository. https://github.com/lxc/lxc. (2018).

[22] Filipe Manco, Costin Lupu, Florian Schmidt, and others. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 218–233.

[23] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014).

[24] Mesosphere. 2019. Marathon. https://github.com/mesosphere/marathon. (2019).

[25] NIST. 2019. NIST report for CVE-2019-5736. https://nvd.nist.gov/vuln/detail/CVE-2019-5736. (2019).

[26] Rami Rosen. 2013. Resource management:Linux kernel Namespaces and cgroups. https://www.cs.ucsb.edu/ rich/class/cs293b-cloud/papers/lxc-namespace.pdf. (2013).

[27] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and others. 2004. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, Berkeley, CA, USA, 16–16.

[28] Mickaël Salaun. 2018. File access-control per container with Landlock *(FOSDEM '18)*. Brussels. https://landlock.io/talks/2018-02-04_landlock-fosdem.pdf

[29] Mickaël Salaün. 2018. Landlock Documentation about administrator rights. https://github.com/landlock-lsm/linux/blob/landlock-v8/Documentation/security/landlock/index.rst. (2018).

[30] Ravi Sandhu. 2013. Access Control Models. https://www.profsandhu.com/cs6393_s13/L2.pdf. (2013).

[31] Michael Schwarz, Samuel Weiser, and Daniel Gruss. 2019. Practical Enclave Malware with Intel SGX. (2019).

[32] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, and others. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 121–135.

[33] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. Implementing SELinux as a Linux security module. *NAI Labs Report* 1 (2001), 43.

[34] M. Souppaya, J. Morello, and K. Scarfon. 2017. Application container security guide. (2017).

[35] Yuqiong Sun, David Safford, Mimi Zohar, and others. 2018. Security Namespace: Making Linux Security Frameworks Available to Containers. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, Berkeley, CA, USA, 1423–1439.

[36] Chris Wright, Crispin Cowan, Stephen Smalley, and others. 2002. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 17–31.